MICROSOFT®

TECHNICAL JOURNAL

Microsoft Confidential

Editor, Ross Garmoe

July 16, 1984
Volume IV

"The greatest gift that God could give us
To see oursel's as others see us
It would from mony a blunder sa'e us"

Robert Burns, from "To a louse,
on seeing one on a lady's bonnet"

EDITOR's COMMENTS - Ross Garmoe

At Microsoft, we pride ourselves on delivering high-quality, state-of-the art software. We point to our industry standard BASIC which is on 90 percent of all microcomputers in the world. MS-DOS is the standard operating system for single-user 16-bit processors, and XENIX is widely used for multiuser operations. Our productivity tools (Multiplan, Word, etc.) and the languages (FORTRAN and Pascal) are consistently among the top items in their class. The SoftCard for the Apple founded the subindustry of imbedded hardware/software extensions for microcomputers.

This consistently high-quality software and hardware has made Microsoft a very influential force in the microcomputer industry. Our software and hardware expertise is much in demand. In many cases, we are consulted by hardware manufacturers during the design phase, which allows us to leverage our expertise. We are currently the largest manufacturer of microcomputer software, and we have the potential to become the dominant manufacturer. The challenge facing us is to not become complacent with our past successes but to use these as a base for doing even better.

Some of the areas in which we need to improve our efforts include:

- Responsiveness to Customer Needs
- Design Methodology
- Implementation Techniques
- Testing and Verification Techniques
- Scheduling

RESPONSIVENESS TO CUSTOMER NEEDS

When a new product or new features are being considered for an existing product, we must be sure that the real needs of the user are being considered. It is easy to solve "almost the right problem" or to add so many features that the product becomes difficult to use or understand. The product should be so natural to use that a few hours of study should allow productive work to be performed. Additional features should be natural extensions of basic concepts. Most of the users are not computer experts and will be scared away if the product appears too complex. "Creeping featurism" should be treated like the plague: avoided if at all possible and treated with large doses of common sense if encountered.

It is important that the product be presented in a manner appropriate to the computer sophistication of the intended users. Spreadsheet packages are successful because they present the power of the computer in the natural paradigm of the rows and columns of an accountant's spreadsheet. New applications must be designed to easily fit into the intended environment. This will be difficult to do unless we have a detailed understanding of the application field. In many cases, this will mean that the product design will include both application and software experts.

DESIGN METHODOLOGY

Design methodology has become an emotional term in recent years. Many authors have presented different methodologies and claimed major improvements in programmer productivity if their system was followed in exhausting detail.

However, design methodology is not a simple matter of choosing HIPO over structured life-cycle or Zen and the Art of Computer-Bit-Fiddling. The important point to remember about all of the design methodologies is that there are tools and techniques that encourage the design of reliable software systems that are easy to maintain and enhance. It is our responsibility to be aware of the advantages and disadvantages of the different techniques and to choose those best suited to the task at hand.

There are many books available on design methodology. As a start, I suggest reading some of the following:

Concise Notes On Software Engineering by Tom De Marco

Managing the System Life Cycle by Edward Yourdan

The Design of Design by Gordon L. Glegg

IMPLEMENTATION TECHNIQUES

Microsoft has made C the language of choice for implementing new software packages, with assembler to be used for those sections that require high performance. This choice of C allows carefully written programs to be portable across most of the interesting CPUs and systems. C is a powerful language, but it is very terse and allows one to easily write very tricky and obscure code, often to the detriment of the task at hand. This, coupled with the tendency for C programmers to use few if any comments, can make reading somebody elses program, or even your own several months later, a challenging puzzle. The point of programming is to generate a product, not to demonstrate mastery of all of the arcane features of the language. When writing a program, choose the simplest algorithm and techniques appropriate to the problem. If an array of pointers to functions returning pointers to struct foo is the best implementation, then use it, but document what is happening! Forcing someone else to waste time trying to understand what a program is supposed to do is a use of personnel time that we cannot afford.

There are many sources of new implementation techniques. In fact, the problem is that there are so many that it is impossible to read even a small fraction and still have time to accomplish any work. The important thing is to try to read at least a few of the sources on a continuing basis. We must continue to study and learn or the advances in the field will quickly make us obsolete. When you find a technique that is especially interesting or useful, let other people know about it. If you have a problem, talk with others about what techniques or references are useful. We are not here to reinvent the wheel. One source that I have found interesting is The Elements of Programming Style by Kernighan and Plauger.

TESTING AND VERIFICATION

Testing and verification is very difficult and if it is done correctly takes a lot of time and is very expensive in people and machine resources. However, the effect of not properly testing a product can be even more expensive. The reputation of Microsoft will be severely damaged if we start distributing products which are poorly tested and have to be recalled. The real cost of not properly testing a program can be much more expensive than testing it.

The viewpoint required by the testing process is different from that required by the design and programming process. Design and programming is a constructive process while testing is a destructive one. Most programmers find it difficult to approach their own programs with the "destructive" frame of mind that is best suited for finding bugs/implementation errors. In many cases, it is helpful to swap some of the testing effort with somebody else. This makes it easier to attack the code and also has the side benefit of allowing you to read and learn from code written by other programmers.

To be done well, the testing process needs to be designed at the same time as the product and with the same attention to detail. Testing and test design is rarely taught in school and most programmers learn it in an ad hoc manner as the need arises. The problems of testing will be addressed further in later journal issues, but in the meantime, I recommend that you read the following books:

The Art of Software Testing by Glenford J. Meyers

Software Testing Techniques by Boris Beizer

SCHEDULING

Figuring out how long a project should take and then accomplishing the work within that time is probably the thing that we all do poorly. An architect can design a building and specify years in advance the date and time that the building will be ready for occupancy. The best that I can do for scheduling is figure out how long I think the work should take, double the estimate and go to the next higher unit. That is why a one week project takes two months.

The microcomputer software industry is moving fast. New products seem to spring out of thin air at a phenomenal rate. Being on time with a product can mean the difference between success and failure. Since much of our work is done in conjunction with other manufacturers, delays affect them as well as Microsoft. If we develop and maintain a reputation for accurate scheduling, these companies will continue to include Microsoft in their product planning. We will then continue to have access to the state-of-the-art hardware/software systems that make our jobs interesting.

Studies are available that model the relationships between the complexity of the programming task, the resources available, and the amount of time required to complete the work. One system that implements such a model is WICOMO which is an interactive software cost estimation model based on the COCOMO model developed at TRW and described in Barry Boehm's book Software Engineering Economics (Prentice-Hall, 1983).

WICOMO calculates estimates of man-months of effort, cost, and scheduling time required for a project based on its size and a number of additional parameters (called "cost drivers"). The input to the model consists of:

- The "Development Mode" (organic, semi-detached, or embedded)
- Values of Each of the 14 Cost Drivers
- The Cost per Man-month of Analyst and Programmer Time
- An Estimate of the Number of Source Instructions

The cost drivers vary from project to project. A manager or software engineer using WIOMO can choose the appropriate values for a given project. The following cost drivers are used in the COCOMO model to derive estimates:

Required Software Reliability	Database Size
Product Complexity	Execution Time Constraint
Virtual Machine Volatility	Computer Turnaround Time
Analyst Capability	Analyst Experience
Programmer Capability	Virtual Machine Experience
Programming Language Experience	Required Implementation Schedule
Use of Software Tools	Use of Modern Programming Practices

We need to evaluate this and other models and determine if they can accurately predict the development times. To do this type of study, we need to keep records on how much effort is spent on projects and how well the end result meets the original design specifications. Only with such evaluation can we begin to accurately predict the amount of time a programming task will require.

An interesting book that explains why things always go wrong is the Mythical Man Month by Fred Brooks. This book is based on his experience as the manager of OS/360 and gives valuable insight into managing large projects.

SOFTWARE CODE SEGMENT TIMING - Mark Bebie

On the MS-NET project, we needed to find where the network was spending its time (where the bottlenecks were), and hopefully uncover some unexpected delays. This was critical not only because of the large volume of software written, but also that the software runs on several machines at once, and this real-time machine interaction is not well defined. So far, the timing we have done has been directed at speeding up the client-file server data exchange. Some of the measurements taken have dealt with setup, disk I/O, DMA vs. copy, queing of network requests, asynchronous command completion, round-trip packet time, all for both sides of a network transaction.

To time the software, we had to find a clock. The system clock provided by the DOS does not have enough resolution. The system clock is driven off the Intel 8253 chip, but the functions provided by that chip did not meet our needs for timing many operations at once. We suggested that Terry Lipscomb in hardware build us a timer card. The card has a 32-bit counter incrementing every 250 nanoseconds. Since we were timing blocks of code on the order of hundreds and thousands of microseconds, we found this resolution too fine, and a one or ten microsecond resolution would be adequate. In order to validate our results, we checked times externally with a wristwatch.

The DOS device handler for this card is a terminate-and-stay-resident program (measure.exe) which hooks an unused interrupt vector (0xBB), and fields interrupts by the timed software. To communicate with measure, interrupt 0xBB is issued along with a function code ("start timing" or "stop timing") and a unique preassigned "handle" (0x00 - 0xFF) identifying the code segment. To time an operation, the code is simply framed with interrupts to "start timing" and to "stop timing," both using the same "handle." Measure keeps a count of the handle usage, the minimum and maximum times, and the running total.

Variance and standard deviation would also be nice to have, but computing them on the fly introduces too much delay and keeping several hundred 32-bit counts for several handles at once runs into space problems.

More than one bug was uncovered with the mesurements we took. A subtle one was a read request in the file server protocol. This small message would take anywhere from 28 ms (milliseconds) to 229 ms to get to the server. After a thorough examination, all the software in the path was doing its job quickly, yet the read request time was still averaging 160 ms. We later found out that our transport layer, written by an outsider, would drop incoming messages if there was no receive posted by the user process. The counters kept by the transport shows 81 messages rejected due to no receive posted out of 100 read requests. After changing the file server software to keep a receive posted, the reject count went to zero and the read request time averaged 30 ms.

The timing we have done has proven to be invaluable in speeding up MS-NET by a factor of three since we started using the timer card.

It enabled us to obtain a concrete profile of the interaction of the network software, and I would highly recommend its use to all software developers.

The timer board has been popular with other groups which has prompted the hardware department to design another card with three times, with pre-selectable resolution down to one microsecond.

A NEW FEATURE IN THE C COMPILER - Ralph Ryan

The newest release of the Microsoft C Compiler (3.00.07) has an important new feature:

> Function declarations will accept the type of the formal argument, and function calls will check the actual arguments against this type. This feature is part of the proposed ANSI-C specification.

One of the most common (and subtle) errors in a C program is illustrated in this example:

```
    ex1()
            {
            int i;
            int j;
            ex2(i, 0, j);
            }
    ex2(a, b, c)
            int a;
            long b;
            int c;
            {
            .
            .
            }
```

On a machine with 16 bit "int" (such as the 8086), there is a mismatch in the arguments. The problem is even more difficult to detect if the functions ex1 and ex2 are in different compilands.

The new declaration syntax is:

```
    [storage class][type]function_name([type [,type ....]]);
```

In the example above, declaring

```
    int ex2(int, long, int);
```

would have caused the compiler to issue warnings about the type problems.

The syntax for argument type declaration is identical to that for types, with two added features:

- a trailing comma denotes a variable length argument list.
- a 'void' denotes 0 arguments.

The compiler checks types as for assignments, and will warn if mismatches occur. Declarations in the 'old style' (i.e., with no argument types) will be processed as before, with no argument checking. Old programs will continue to compile without modification. Even if you do not declare functions using this feature, the compiler will issue warnings if you have defined a function and used it in an incompatible manner later in the same source file.

A few more examples:

```
    extern int foo(void);           /* check for 0 arguments */
    extern int bar(int, char*);     /* one int, one char pointer */
    extern int zot();               /* old-style -- no checking */
    extern printf(char *,);         /* check first arg, ignore rest */
```

Note that the compiler does NOT correct the error -- it will only warn.

The best way to use this feature is to include a header file that declares the types of all function arguments. Within the next few weeks a tool will be available to automatically generate such a header file from a set of source files. We will also make available a header file with argument declarations for all of the C library functions.

MICROSOFT C (cmerge) VS. LATTICE - Hans Spillar and Ralph Ryan

The following is a set of differences between the Lattice C compiler and the Microsoft "standard" definition of C (MSC). The numbers correspond to the appropriate section of Kensington a Ritchie, The C Programming Language.

2.1:
In MSC, comments do not nest. Lattice provides this under switch control, but the default is that they do nest.

2.1:
MSC allows A-Z, a-z, and _ as the first character in an identifier, and those plus 0-9 for subsequent characters. Lattice allows '$' for subsequent characters too.

2.1
MSC allows macro declarations to be multiple lines, with the continuation indicated by a '/' before the new line. Lattice does not allow for the continuation.

2.5:
MSC strings are unique. They can be used as initializations and so forth, and can be modified at runtime. Lattice detects when the initialized strings are the same and generates only on instance.

2.5:
MSC does not allow multi-character constants. Lattice allows them.

2.5:
The current version of Latice does not do structure assigns, passes, and returns. Current versions of Lattice, when given the name of a structure, pass the address of the structure.

2.5:
While the C standard says that char can be either unsigned or signed, a great many C programs rely on char being signed. As a result, Microsoft has taken the position that the char type should be signed, and an unsigned char type is provided. Additionally, unsigned long has been provided.

2.5:
An array or procedure name is an address -- a constant pointer to a thing of the given type. Lattice seems to think that there is an extra dereference involved, and requires an '&' to get the address. A consequence of this is that if the name of an array is given where an address is expected, the first two bytes of the array are given as the pointer.

2.5:
Casting a value to a pointer type produces an lvalue. Lattice does not consider the result of any cast an lvalue. The following is a legal (if unuseful and potentially buggy) MSC routine.

```
main()
{
        char *p;
        short l;
        *((long *)p)++ = 47;
        *p = *(char *)& l;
}
```

2.5:
The current draft standard has improved the preprocessor a little:

* #elif ... has been added.
* defined(...) has been added. #if defined(FOO) is the same as #ifdef FOO.
* preprocessor directives can appear as the first non whitespace on a line. They needn't begin with the first character on a line.
* &&, ::, etc. were omitted from the white book defintion through an oversight.

MSC supports all of these.

7.6/7:
Lattice types equality operators according to the left hand side, rather than as the maximum of the arguments.

8.1:
Lattice allocates bitfields high order to low order on the 8086, where MSC goes low to high. In an expression such as

```
(pointer expression)->bitfield += expression;
```

Lattice evaluates the pointer twice (as p->b = p->b + expression);
This is incorrect when there are side effects in the pointer expression.

8.1:
It is a common practice in C to check for array limits by comparing against the dimension of the array:

```
int a[MAX];
prac() {
        int *p;
        if (p< &a[MAX]) ....
```

Lattice complains about exceeding the bounds, even though no memory reference has taken place. Oddly, Lattice does not complain about p > a[-1].

8.1:
Lattice doesn't seem to know about enumerations at all. This is in the v7 language and all subsequent versions.

8.1:
A static variable with no explicit storage class or initialization is allocated at compile time in Lattice, whereas in UNIX C and MSC it is treated as an undefined variable with a size, which will either defer to a separate declaration of be 'communally allocated' by the linker. Current versions of the Microsoft 8086 linker will allocate such undefined symbols.

MASM86 - Ross Garmoe

Version 3.04 of the 8086 assembler is now available for MS-DOS and the 68K XENIX systems. The MS-DOS version is available on a floppy disk on Reuben Borman's door. The 68K version is available on all 68K systems as /usr/tools/masm and /usr/tools/masm86. After the first of August, the assembler will be accessible only as /usr/tools/masm86. This name change is to make all of the 8086 tools conform to the same naming conventions and also to allow masm style assemblers for other chips to reside in /usr/tools.

In addition to bug fixes, version 3.04 provides several new features. A case sensitivity switch has been added to allow PUBLIC/EXTERN items to be in mixed case or to allow all symbols to be case sensitive. The MS-DOS version can now assemble much larger programs because macro and symbol text is kept in a separate segment. The MS-DOS verbose statistics (see below) display the amount of symbol and string space free at the end of the assembly.

On the 68K systems, the relevant files are:

/usr/tools/masm	the assembler (goes away August 1)
/usr/tools/masm86	the assembler
/usr/tools/masm86.doc	the command line options and exit codes
/usr/tools/V3.04_README	listing of bugs corrected in this version

On the floppy disk, the relevant files are:

masm300.exe	the assembler
masm300.doc	the command line options and exit codes
README	listing of bugs corrected in this version

The current assembler command format is as follows:

```
masm86 [options] filename
```

The options are toggles with the following meanings and default settings.

flag	default	meaning of TRUE condition
a	FALSE	output segments in alphabetic order
c	FALSE	output cross reference data to filename.crf
d	FALSE	pass 1 listing to filename.lst
e	FALSE	use floating point emulation
Ipath	NULL	prefix for search path for include files; up to 10 include paths allowed
lfname	FALSE	listing (FALSE overrides d) to fname. If no fname is specified, listing is written to filename.lst. If fname is -, listing is written to stdout.
Mc	Mu	set case sensitivity switch u - map all symbols to upper case x - map all symbols except PUBLIC and EXTERN to upper case l - leave symbol case alone
n	TRUE	output symbols if -l selected
O	FALSE	output listing in octal

o	TRUE	output binary
r	TRUE	use real 8087 instead of Microsoft format
v	FALSE	print verbose statistics on console if not selected, only error counts will be displayed if assemble errors
X	FALSE	toggle setting of conditional listing flag
x	TRUE	list errors to console

Exit codes have the following meanings

code	meaning
0	no error
1	argument error
2	unable to open input file
3	unable to open listing file
4	unable to open object file
5	unable to open cross reference file
6	unable to open include file
7	assembly errors
8	if there are fatal assembly errors, the object file is deleted
9	real number input not allowed in this version

* * * *